Parallel Distributed Memory Programming - Message Passing

(How to efficiently parallelize your code with great difficulty)

Tom Logan
ARSC

Overview

I. Background
II. Introduction to MPI
III. Basic MPI Programs
IV. Intermediate MPI Concepts
V. Conclusions

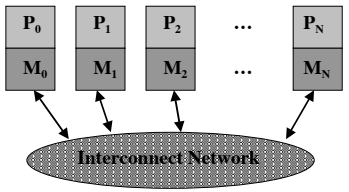
Section I: Background

Section Contents

- Distributed Memory Architectures
- Architecture Key Features
- Message Passing
- Message Passing Benefits

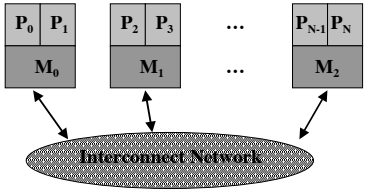
Distributed Memory

- **Each node contains one processor with its own memory.**
- **Nodes are interconnected by message passing network (switches or routers)**



Distributed Shared Memory

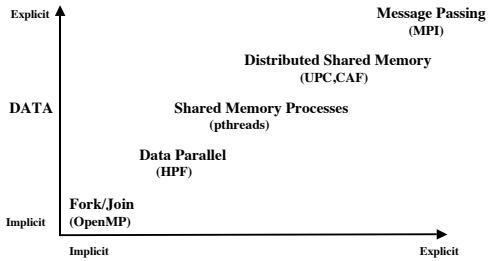
- **Modest number of processors share memory within a single node**
- **Distributed memory among nodes**
- **Examples include the IBM p655 & p690 and the SUN x2200 & x4600 nodes**



Architecture Key Features

- **Memory Latency**
  - Remote memory access is high latency
  - May be possible only with message passing
- **Network Latency**
  - Time to receive a message from another node
  - Several orders of magnitude slower than memory
- **Bandwidth**
  - Measure of the throughput of interconnect
  - 10's-100's MB/s inter-node
  - 10's-100's GB/s intra-node

Parallel Models & Languages



(Figure Copied From Cray Training Manual)

Shared Memory

- **Implicit Parallelism**
  - Insert directives only into code
  - Compiler distributes work
  - Compiler distributes data
  - Explicit synchronization

Message Passing

- **Explicit Parallelism**
  - Insert communication calls "manually"
  - Messages consist of status & usually data
  - Includes both point-to-point and global communications
  - NO Remote Memory Access is provided

Message Passing Benefits

- **100% control over data distribution**
- **100% control over work distribution**
- **Better performance than other models**
- **Provides excellent scalability**
- **Provides excellent locality**
- **Portable standard exists**

Section II: Introduction to MPI

Section Contents

- **What's MPI**
- **Getting Started**
- **Groups and Ranks**
- **Running and Compiling**
- **Hello World Example**

What's MPI?

- **Standard for Distributed Memory Programming**
  - Standard has existed for over 20 years
  - Provides source code portability across many platforms
  - Has 300+ routines defined in MPI2 API
  - Only need 1/2 dozen of them to write meaningful code

Getting Started

- **Header Files**
  - Fortran include 'mpif.h'
  - C #include <mpi.h>
- **Start Up and Shut Down**

```
CALL MPI_INIT(istat)
…
CALL MPI_FINALIZE(istat)
```

- **Routine format**
  - Fortran CALL MPI_xxx(…,istat)
  - C int stat=MPI_Xxx(…);

Groups and Ranks

- **Communicator**
  - MPI grouping of PEs
  - Default is named MPI_COMM_WORLD
  - Routine MPI_Comm_Size(…) gets NPES
- **Rank**
  - PE number; zero based
  - Routine MPI_Comm_Rank(…) gets MYPE

Compiling and Running

- **Compilation**
  - Often need to add a library during linking (typically '-lmpi')
  - On current ARSC systems call special compiler instead
- **Running**
  - Most systems use the 'mpirun' command
    mpirun -np <NPES> <USER_EXE>
  - Example
    mpirun -np 4 ./hello
- **For full details on ARSC machines see**
  http://www.arsc.edu/arsc/support/howtos/usingpacman/

Example - Hello World

```
program hello
include 'mpif.h'
integer istat, ierr, npes, mype

call mpi_init(istat)
call mpi_comm_size(mpi_comm_world,npes,ierr)
call mpi_comm_rank(mpi_comm_world,mype,ierr)
print *,' Hello world from ',mype,' of ',npes
call mpi_finalize(ierr)

end
```

```
#include <stdio.h>
#include <mpi.h>

int main(int argc,char** argv)
{
  int mype,npes;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&npes);
  MPI_Comm_rank(MPI_COMM_WORLD,&mype);
  printf("Hello world from %i of %i\n",mype,npes);

  MPI_Finalize();
}
```


Hello World - Output

```
mg56: 97% mpif90 -o hello hello.f90
mg56: 98% qsub test.pbs
553506.mpbs1
mg56: 99% cat test.pbs.o553506
Wed Nov  5 16:43:23 AKST 2008
 Hello world from 0 of 4
 Hello world from 3 of 4
 Hello world from 2 of 4
 Hello world from 1 of 4
Cleaning up all processes ...
done.
Wed Nov  5 16:43:24 AKST 2008
```


Torque Script - MPI Example

```
#!/bin/bash
#PBS -q standard
#PBS -l walltime=8:00:00
#PBS -l nodes=4:ppn=12
#PBS -j oe

cd $PBS_O_WORKDIR

mpirun –np 48 ./myprog
```

Section III:
Basic MPI Programs

Section Contents

MPI Blocking Send

Fortran: MPI_SEND(buf, cnt, dt, dest, tag, comm, ierror)
<type> BUF(*), INTEGER count, datatype, dest, tag, comm,ierror
C: int MPI_Send(void *buf, int cnt, MPI_Datatype dt, int dest, int tag, MPI_Comm comm);

buf	initial address of the send buffer
cnt	number of elements in the send buffer (nonnegative integer)
dt	data type of each send buffer element (handle)
dest	rank of the destination (integer)
tag	message tag (integer)
comm	communicator (handle)
ierror	return code value. Returns MPI_SUCCESS upon successful completion. MPI_SUCCESS is defined in mpif.h

MPI Blocking Receive

Fortran: MPI_RECV(buf, cnt, dt, src, tag, comm, st, ierror)
<type> BUF(*), INTEGER cnt, dt, src, tag, comm, st(MPI_STATUS_SIZE), ierror
C: int MPI_Recv(void *buf, int cnt, MPI_Datatype dt, int src, int tag, MPI_Comm comm, MPI_Status *st);

buf	initial address of the receive buffer	
cnt	number of elements in the receive buffer (nonnegative integer)	
dt	data type of each receive buffer element (handle)	
src	rank of the source (integer)	{MPI_ANY_SOURCE is wildcard}
tag	message tag (integer)	{MPI_ANY_TAG is wildcard}
comm	communicator (handle)	
st	returns the status object	
ierror	return code value. Returns MPI_SUCCESS upon successful completion. MPI_SUCCESS is defined in mpif.h	

MPI Elementary Data Types

Fortran	MPI
INTEGER	MPI_INTEGER
REAL	MPI_REAL
DOUBLE PRECISION	MPI_DOUBLE_PRECISION
COMPLEX	MPI_COMPLEX
LOGICAL	MPI_LOGICAL
CHARACTER(1)	MPI_CHARACTER
…	…
DOUBLE COMPLEX	*MPI_DOUBLE_COMPLEX*
*REAL*8*	*MPI_REAL8*
*INTEGER*2*	*MPI_INTEGER2*

Example - blocking.f90

```
program blocking
    include 'mpif.h'
    parameter (n=1000)
    integer mype, npes, serr, rerr
    integer status(mpi_status_size)
    integer, dimension(n) :: sbuf, rbuf

    call mpi_init(ierr)
    if (ierr.ne.MPI_SUCCESS) stop 'bad init'
    call mpi_comm_size(MPI_COMM_WORLD,npes,ierr)
    if (npes.ne.2) stop 'must have npes = 2'
    if (ierr.ne.MPI_SUCCESS) stop 'bad size'
    call mpi_comm_rank(MPI_COMM_WORLD,mype,ierr)
    if (ierr.ne.MPI_SUCCESS) stop 'bad rank'
    iflag=1
    do i=1,n
        sbuf(i)=i
    end do
```


Example - blocking.f90

```
if (mype.eq.0) then
    call mpi_send(sbuf,n,MPI_REAL,1,99,MPI_COMM_WORLD,serr)
    if (serr.ne.MPI_SUCCESS) stop 'bad 0 send'
    call mpi_recv(rbuf,n,MPI_REAL,1,98,MPI_COMM_WORLD,status,rerr)
    if (rerr.ne.MPI_SUCCESS) stop 'bad 0 recv'
else
    call mpi_recv(rbuf,n,MPI_REAL,0,99,MPI_COMM_WORLD,status,rerr)
    if (rerr.ne.MPI_SUCCESS) stop 'bad 1 recv'
    call mpi_send(sbuf,n,MPI_REAL,0,98,MPI_COMM_WORLD,serr)
    if (serr.ne.MPI_SUCCESS) stop 'bad 1 send'
end if
do j=1,n
    if (rbuf(j).ne.sbuf(j)) iflag=0
    if (rbuf(j).ne.sbuf(j)) print *,' process ',mype,' :rbuf( ',j,' )= ',rbuf(j),' - should be ',sbuf(j)
end do
if (iflag.eq.1) then
    print *,' Test passed on process ',mype
else
    print *,' Test failed on process ',mype
end if
call mpi_finalize(ierr)
end
```


MPI_Sendrecv

- **Shorthand for both sending AND receiving messages in a single call**

  MPI_Sendrecv(sbuf, scnt, sdt, dest, stag,
  rbuf, rcnt, rdt, src, rtag,
  comm, status, ierror)

- **1st 5 parameters are for the send**
- **2nd 5 parameters are for the receive**
- **Also includes communicator, receive status array, and error return**

MPI_Sendrecv Example

- **Blocking.f90 data transfer would look like this with MPI_Sendrecv**

```
if (mype.eq.0) then
    call mpi_sendrecv(      sbuf,n,MPI_REAL,1,99,           &
                            rbuf,n,MPI_REAL,1,98,MPI_COMM_WORLD,status,rerr)
    if (rerr.ne.MPI_SUCCESS) stop 'bad 0 send/recv'
else
    call mpi_sendrecv(      sbuf,n,MPI_REAL,0,98,           &
                            rbuf,n,MPI_REAL,0,99,MPI_COMM_WORLD,status,rerr)
    if (rerr.ne.MPI_SUCCESS) stop 'bad 1 send/recv'
end if
```


Asynchronous Messages

- **Sending a message**

CALL MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierr)

- **Requesting/Checking for a message**

CALL MPI_IRECV(buf, count, datatype, src, tag, comm, request, ierr)
CALL MPI_IPROBE(src, tag, comm, flag, stat(MPI_STATUS_SIZE), ierr)

- **Completion of message request**

CALL MPI_WAIT(request, stat(MPI_STATUS_SIZE), ierr)
CALL MPI_TEST(request, flag, status(MPI_STATUS_SIZE), ierror)

- **Data types for new parameters**

LOGICAL flag
INTEGER source, request, stat(MPI_STATUS_SIZE)

Example - asynch_mpi.f

```
SUBROUTINE asynch(n,sbuf,to,stag,rbuf,from,rtag)
    include 'mpif.h'
    integer, intent(in)  : : from, to, rtag, stag
    integer              : : serr, rerr, rreq, sreq, status(mpi_status_size)
    real, intent(in)     : : sbuf(n)
    real, intent(out)    : : rbuf(n)

    call mpi_irecv(rbuf,n,MPI_REAL,from,rtag,MPI_COMM_WORLD,rreq,rerr)
    call mpi_isend(sbuf,n,MPI_REAL,to,stag,MPI_COMM_WORLD,sreq,serr)

    ! Opportunity to do some work in here…

    call mpi_wait(rreq,status,rerr)
    call mpi_wait(sreq,status,serr)
    if (rerr.ne.MPI_SUCCESS.or.serr.ne.MPI_SUCCESS) stop 'bad communication'
    end
```


Section IV: Intermediate MPI Concepts

Section Contents

- **Collectives**
- **User Defined Data types**
- **User Defined Communicators**

Collectives

- **Data Movement Functions** - broadcasting, scattering, and gathering
- **Global reductions -** application of a binary operator to reduce data (often to a single result)
- **Barrier -** synchronization point

Broadcast

MPI_Bcast(ibuf,icnt,itype,root,comm)

- Sends data from root PE to all others in communicator, e.g. metadata read from a file
- Data of type itype, size icnt, are sent from ibuf on root PE to ibuf on all other PEs

Gather

MPI_Gather(ibuf, icnt, itype, obuf, ocnt, otype, root, comm)

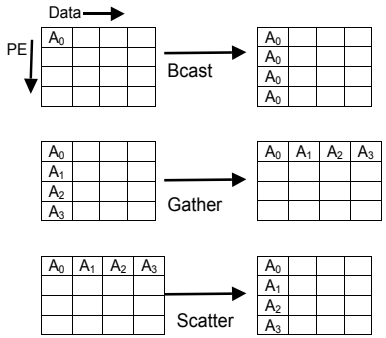
- Sequentially assembles data from a group of tasks into the output buffer on the root PE.
- Data of type itype, size icnt, are sent from ibuf by each PE
- Data are gathered into obuf, which is size ocnt*NPES and type otype, on the root PE. No other PE gets results.
  - obuf must be npes*ibuf in size on the root PE
  - obuf is ignored on all other PEs

Scatter

MPI_Scatter(ibuf, icnt, itype, obuf, ocnt, otype, root, comm)

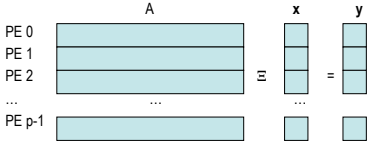
- Scatters data from a single task to a group of tasks such that each gets an equal sequential portion
- Data of type itype, size icnt, sent from ibuf on the root PE
- Data are scattered into obuf, which is size ocnt and type otype, on all PEs.
  - ibuf must be npes*obuf in size on the root PE
  - ibuf is ignored on all other PEs

Collective Data Movements



Example: Matrix Vector Product

We would like to multiply matrix A of size m x n times vector **x** to create vector y which is to be written out. The matrix and vectors are distributed in a ***block-row*** distribution wherein each of p processors has m/p rows of the matrix and the corresponding sections of the vectors. Graphically:



Example: Matrix Vector Product

```
Serial_Mat_Vec_Prod (A,m,n,x,y)
integer, intent(in)      :: m,n
real, intent(in)         :: A(m,n),x(n)
real, intent(out)        :: y(m)
integer                  :: k,j
do k=1,m
  y(k)=0.0
  do j=1,n
     y(k)=y(k)+A(k,j)*x(j)
  end do
end do
end
```

```
Parallel_Mat_Vec_Prod (A,m,n,p,x,y,global_y)
integer, intent(in)      :: m,n,p
real, intent(in)         :: A(m/p,n),x(n/p)
real, intent(out)        :: y(m/p), global_y(m)
integer                  :: k,j,ierr
real                     :: global_x(n)
Call MPI_Allgather(x, n/p, mpi_real, global_x, n/p, mpi_real,
      mpi_comm_world, ierr)
do k=1,m/p
  y(k)=0.0
  do j=1,n
     y(k)=y(k)+A(k,j)*global_x(j)
  end do
end do
Call MPI_Gather(y, m/p, mpi_real, global_y, m/p, mpi_real, 0,
      mpi_comm_world,ierr)
end
```

Global Reductions

MPI_Reduce(sbuf,rbuf,cnt,dt,OP,root,comm,st)
MPI_Allreduce(sbuf,rbuf,cnt,dt,OP,comm,st)

- Lots of operations available
  - MPI_MAX, MPI_MAXLOC
  - MPI_MIN, MPI_MINLOC
  - MPI_SUM, MPI_PROD
  - MPI_LAND, MPI_LOR, MPI_LXOR
  - MPI_BAND, MPI_BOR, MPI_BXOR

Example: Global Sum 1

```
program test_sum
include 'mpif.h'
integer mype,npes,serr,rerr,n,i,j,begin,end,ierr
real     my_sum, sum, Global_Sum
parameter   (n=1000)
real, dimension(n)          :: sbuf
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,npes,ierr)
call mpi_comm_rank(mpi_comm_world,mype,ierr)
do i=1,n
      sbuf(i)=i
end do
begin = mype * n/npes + 1
end   = (mype+1)*n/npes
if (mype.eq.npes-1)  end = n
print *,'PE ',mype,'adding ',begin,'to',end
my_sum=0
sum=0
do j=begin, end
      my_sum = my_sum + sbuf(j)
end do
sum = Global_Sum(mype,npes,my_sum)
print *,'PE ',mype,'Global sum is ',sum
call mpi_finalize(ierr)

end
```


Example: Global Sum 1

```
Real Function Global_Sum(mype,npes,my_sum)
include 'mpif.h'
integer,intent(in)          :: mype, npes
real, intent(in)            :: my_sum
integer                     :: ierr, sender, status(mpi_status_size)
real                        :: sum, tmp
if (mype.ne.0) then
      call mpi_send(my_sum,1,MPI_REAL,0,mype,MPI_COMM_WORLD,ierr)
else
      sum = my_sum
      do sender=1,npes-1
            call mpi_recv(tmp,1,MPI_REAL,sender,sender,MPI_COMM_WORLD,status,ierr)
            sum = sum + tmp
      end do
      Global_Sum = sum
end if
end
```


Output Global_Sum.f90

```
mg56: 4% mpif90 -o global_sum global_sum.f90
mg56: 9% qsub global_sum.pbs
mg56: 15% cat global_sum.pbs.o553514
 PE  0 adding  1 to 250
 PE  3 adding  751 to 1000
 PE  2 adding  501 to 750
 PE  3 Global sum is  1.77964905E-43
 PE  1 adding  251 to 500
 PE  2 Global sum is  1.77964905E-43
 PE  1 Global sum is  1.77964905E-43
 PE  0 Global sum is  500500.
```

Example: Global Sum 2

```
program test_sum
include 'mpif.h'
integer       mype,npes,serr,rerr,n,i,j,begin,end,ierr
integer       status(mpi_status_size)
real          my_sum, sum
parameter   (n=1000)
real, dimension(n)          :: sbuf
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,npes,ierr)
call mpi_comm_rank(mpi_comm_world,mype,ierr)
do i=1,n
      sbuf(i)=i
end do
begin = mype * n/npes + 1
end   = (mype+1)*n/npes
if (mype.eq.npes-1)  end = n
print *,'PE ',mype,'adding ',begin,'to',end
my_sum=0
sum=0
do j=begin, end
      my_sum = my_sum + sbuf(j)
end do
call MPI_Reduce(my_sum,sum,1,mpi_real,
     mpi_sum,0,mpi_comm_world,status)
print *,'PE ',mype,'Global sum is ',sum
call mpi_finalize(ierr)

end
```


Output Global_Sum2.f90

```
mg56: 5% mpif90 -o global_sum2 global_sum2.f90
mg56: 9% qsub global_sum2.pbs
mg56: 15% cat global_sum2.pbs.o553519
PE  0 adding  1 to 250
 PE  1 adding  251 to 500
 PE  2 adding  501 to 750
 PE  3 adding  751 to 1000
 PE  1 Global sum is  0.0000000000E+00
 PE  3 Global sum is  0.0000000000E+00
 PE  0 Global sum is  500500.0000
 PE  2 Global sum is  0.0000000000E+00
```


User Defined Datatypes

- **User can define new MPI types**
  - Allows for messages with mixed data types
  - Allows noncontiguous areas in memory to be specified; makes MPI do the work of packing the data
  - Particularly nice for code readability
- **Routines are named MPI_Type_***

Creating a type

- **Lots of different type functions**
  - MPI_Type_contiguous(count,oldtype,newtype,ierr)
    ex: MPI_Type_contiguous(2,MPI_Real,MPI_TRI)
  - MPI_Type_vector(count, blklen, stride, oldtype, newtype, ierror)
    ex: MPI_Type_vector(xlen,1,ylen,MPI_Real,MPI_ROW)
  - MPI_Type_struct (count, array_of_blocklengths(*),array_of_displacements(*), array_of_types(*),newtype, ierror)
- **After creating the type, it must be committed**
  - MPI_Type_commit(newtype,ierror)

Using the type

```
Subroutine exchang2d(a,sx,ex,sy,ey,MPI_ROW,nbrleft,nbrright,nbrtop,nbrbottom)
use mpi
integer :: sx,ex,sy,ey,MPI_ROW,nbrleft,nbrright,nbrtop,nbrbottom,ierr,nx,stat(mpi_status_size)
double precision :: a(sy-1:ey+1,sx-1:ex+1)

ny = ey - sy + 1

mpi_sendrecv(a(sy,sx),ny,mpi_double_precision,nbrleft,0,
             a(sy,ex+1),ny,mpi_double_precision,nbrright,0, mpi_comm_world,stat,ierr)
mpi_sendrecv(a(sy,ex),ny,mpi_double_precision,nbrright,1,
             a(sy,sx-1),ny,mpi_double_precision,nbrleft,1, mpi_comm_world,stat,ierr)
mpi_sendrecv(a(sy,sx),1,MPI_ROW,nbrtop,2,
             a(ey+1,sx),1,MPI_ROW,nbrbottom,2, mpi_comm_world,stat,ierr)
mpi_sendrecv(a(ey,sx),1,MPI_ROW,nbrbottom,3,
             a(sy-1,sx),1,MPI_ROW,nbrtop,3, mpi_comm_world,stat,ierr)

end subroutine
```


Intracommunicators

- **MPI Intracommunicator - a group of processes that are allowed to communicate with each other and a CONTEXT**
  - Context is system-wide ID tag that marks the groups of processes
  - Context allows a process to be in two intracommunicators with out crosstalk
- **Typical example is MPI_COMM_WORLD**

Intercommunicators

- **MPI Intercommunicator - allows any process in one intracommunicator to transfer data with any process in a different intracommunicator**
- **To create a new communicator use:**

  MPI_Comm_split(oldcom,color,key,newcom)

  - color: all processes with same color put in same newcom
  - key: gives the ordering of new ranks in newcom

Example: MPI_Comm_split

```
program intracom
implicit none
include 'mpif.h'
integer newcom, world_rank, my_rank, my_size, color, ierr

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_Comm_World,world_rank,ierr)
color = log10(real(world_rank+1))/log10(2.)
call MPI_Comm_split(MPI_Comm_World,color,world_rank,newcom,ierr)
call MPI_Comm_rank(newcom,my_rank,ierr)
call MPI_Comm_size(newcom,my_size,ierr)
print *,'world_rank=',world_rank,' comm_rank=',my_rank,' color=',color,' size=',my_size
call MPI_Finalize(ierr)

end
```

Intracom Output

```
mg56: 140% mpif90 -o comm_test comm_test.f90
mg56: 141% qsub comm_test.pbs
mg56: 142% more comm_test.pbs.o559170
 world_rank= 0  comm_rank= 0  color= 0  size= 1
 world_rank= 3  comm_rank= 0  color= 2  size= 4
 world_rank= 7  comm_rank= 0  color= 3  size= 1
 world_rank= 4  comm_rank= 1  color= 2  size= 4
 world_rank= 5  comm_rank= 2  color= 2  size= 4
 world_rank= 6  comm_rank= 3  color= 2  size= 4
 world_rank= 1  comm_rank= 0  color= 1  size= 2
 world_rank= 2  comm_rank= 1  color= 1  size= 2
```


Intercomminucator Creation

Call MPI_INTERCOMM_CREATE(local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror)

INTEGER local_comm, local_leader, peer_comm, remote_leader, tag, newintercomm, ierror

Example: Intercomm

```
program inter
include 'mpif.h'
integer newcom, world_rank, com_rank, my_size, color, ierr, status(MPI_STATUS_SIZE)
real    x
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_Comm_World,world_rank,ierr)
color = mod(world_rank,2)
call MPI_Comm_split(MPI_Comm_World,color,world_rank,newcom,ierr)
If (color.eq.0) call MPI_Intercomm_create(newcom,0,mpi_comm_world,1,9,com,ierr)
If (color.eq.1) call MPI_Intercomm_create(newcom,0,mpi_comm_world,0,9,com,ierr)
call MPI_Comm_rank(newcom,com_rank)
If (color.eq.0) then
      x = 3.56+com_rank
      MPI_Send(x,1,mpi_double,com_rank,32+com_rank,com,ierr)
else
      MPI_Recv(x,1,mpi_double,com_rank,MPI_ANY_TAG,com,status,ierr)
      print *,' G:' ,color,'  P:' ,com_rank,'  tag:' ,status(MPI_TAG)
      print *,' G:' ,color,'  P:' ,com_rank,'  x = ' ,x
end if
call MPI_Finalize(ierr)
end
```


Intercom Output

```
f1n2 55% ./intercomm_test
 G: 1  P: 1  tag: 33
 G: 1  P: 0  tag: 32
 G: 1  P: 3  tag: 35
 G: 1  P: 2  tag: 34
 G: 1  P: 3   x = 6.55999994
 G: 1  P: 2   x = 5.55999994
 G: 1  P: 0   x = 3.55999994
 G: 1  P: 1   x = 4.55999994
```

Advanced MPI Features

- **Other Features Not Covered Here**
  - Topology Functions
  - Derived datatypes
  - Buffered messages
  - Return receipt requested messages

Suggestions for MPI

- **Blocking calls are over used**
  - Try to use non-blocking calls and synchronization
- **Barrier and Waits are often over used**
  - Avoid unnecessary barriers
- **Use MPI routines instead of writing your own**
- **Use few large messages rather than many small ones**
- **Use meaningful message tags**
- **Use specific receives**
- **Interleave work and I/O whenever possible**